Architectural Tradeoffs in Synthesis of Pipelined Controls

Loganath Ramachandran
Daniel D. Gajski

Technical Report #92-49
May 21, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

ramachan@ics.uci.edu

Abstract

Many high level synthesis systems produce designs without any consideration for the underlying architecture. In such systems, tradeoffs between area and delay can only be achieved by changing the synthesis constraints (e.g., number of functional units). These systems do not exploit the wider range of tradeoffs that can be achieved by modifying the underlying architecture. In this report we derive a relationship between architectural constraints and scheduling algorithms, and demonstrate how architectural styles impose certain restrictions on the scheduling process. In particular, we consider different control pipelining architectures. We also propose a versatile scheduling algorithm that is capable of synthesising designs for different control pipelining styles.

Contents

List of Figures

1 Introduction

High Level Synthesis consists of automatically synthesizing hardware from a given abstract high level description. Some of the steps during the synthesis process include: allocation of sufficient number of functional units, scheduling of operations into various control steps and binding the operations and interconnects into the appropriate units.

There have been many research efforts in the areas of scheduling, allocation and binding, that have resulted in powerful algorithms for these tasks [1, 5, 7, 8, 9, 10, 11]. Most of these systems accept either unit constraints or performance constraints during synthesis. Some of the other algorithms [17], minimize the register to register delays to satisfy the given clock constraint.

However the above algorithms do not consider the strong relationship that exists between the underlying architecture and synthesis methods. Consequently the results produced by the above high level synthesis methods, may be good for some architecture styles, but may be unimplementable in other architectures. As an example, the algorithm by Kim et. al. [9] minimizes the number of states in the design by sharing operations from mutually exclusive branches. Although this algorithm reduces the total number of states in the design, it may be impossible to implement this schedule unless we have an architecture that provides for registers on all status signals. The main problem with such above approaches is that the algorithms do not take into account any architectural considerations.

In this report, we derive a relationship between scheduling algorithms and architectural styles. In particular, we concentrate on the various control pipelining strategies and show how scheduling algorithms must perform differently for each control pipelining strategy. This implies that a scheduling algorithm cannot ignore the architecture in which the design would get implemented. We propose a scheduling algorithm, that is driven by the specified control pipelining scheme. We believe that defining such relationships between the architecture and the synthesis algorithms would result in more practical and useful designs.

With the proposed algorithm, it is possible to synthesize efficient designs for a given specific control pipelined architecture. Since control pipelining architectures are mainly intended for performance vs cost tradeoffs, the system automatically exploits the tradeoffs available at the architectural level. We consider all the control pipelining approaches discussed in [16]. An overview of the proposed approach for synthesis with architectural considerations is shown in Figure 1(b).

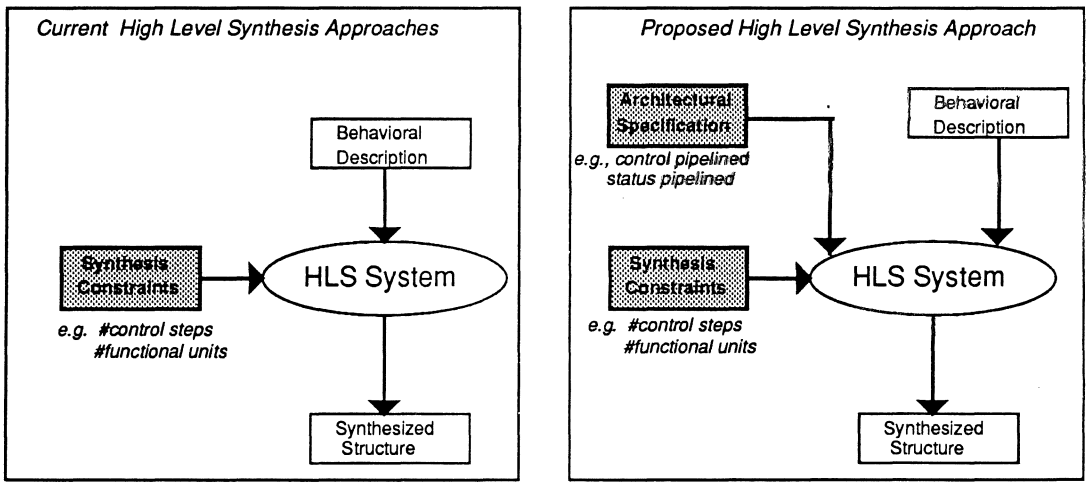

Figure 1: Proposed High Level Synthesis Approach

The rest of the report is organized as follows. In the next section we provide the details of the three architectures that we will be using for the rest of the report. In section three, we discuss how the architectural constraints impact the scheduling phase in high level synthesis. In section four, we show a scheduling algorithm that can efficiently handle all the three architectural styles. Finally we provide the results of our algorithm on a number of standard HLS benchmarks and show how the results are influenced by the selected architectural style.

2 Control Pipelined Architectures

Typically many synthesis systems are targeted for a general CU-DP based architecture. This architecture can be abstractly modeled as a FSMD[4], which is a finite-state machine with a datapath. The model consists of two important parts. The *control part* which can be modeled as a finite state machine consisting of a state register and combinational logic to compute the next-state values and control signals for the datapath. The *datapath* contains the functional units and storage units to perform the required computations.

A sample FSMD is shown in Figure 2(a). The control unit is shown as a 3-state FSM. This control unit communicates with the datapath using *status* and *control* signals. The *control signals* are set by the control unit and used to control the operations of various components in the datapath. On the other hand, the *status signals* are set by the datapath to indicate the status of various computations it has performed.

Each state of this FSMD model can be further subdivided into three micro-actions. These microactions are shown in Figure 2(b). Since each state of the sample state machine can be viewed as three sequential microactions, we can assume that given a state s_i, the machine actually performs three microactions $s_{i_{set\_cntrl}}$, $s_{i_{exec\_dp}}$ and $s_{i_{next\_state}}$. In Figure 2(c), we show all the microactions that are performed in our sample FSMD. Since we have three states in our machine and each state consists of three micro-actions, we have nine different microactions represented as $s_{1_a}, s_{1_b}, s_{1_c}, s_{2_a}, s_{2_b}, s_{2_c}, s_{3_a}, s_{3_b} and s_{3_c}$.

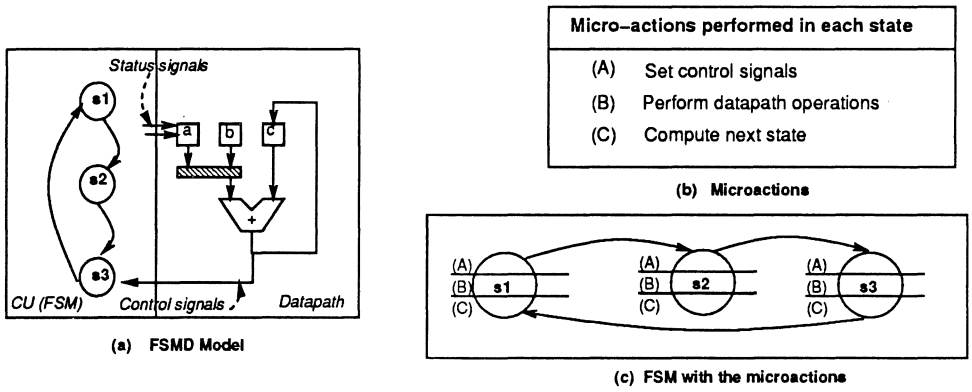

Figure 2: FSMD model

In each state, it is possible to execute these three micro-actions in a pipelined fashion. We refer to this pipelining methodology as *control pipelining* since we are actually pipelining the micro-actions in a state. We now consider three different *control pipelining* methodologies.

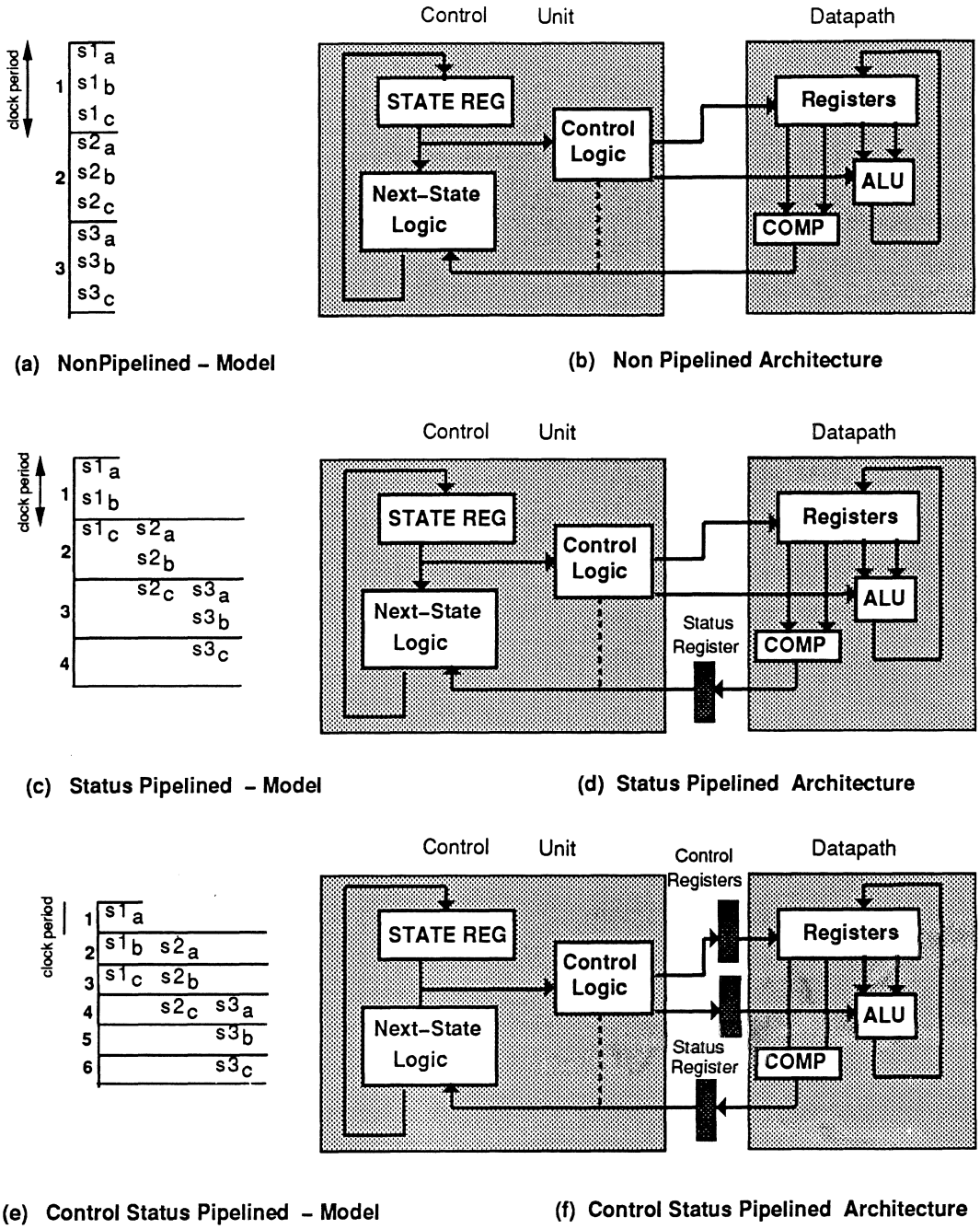

Figure 3: Control Pipelining

2.1 Non-pipelined methodology

In this model, the three microactions in a state are performed serially. In Figure 3(a) we show the execution of the three states in our FSMD. Since all the three microactions in a state are executed sequentially, the min clock period for this architecture can be computed as:

$$Min\_Clk\_Period = t_{control} + t_{datapath} + t_{nxt\_st\_logic} + t_{state\_register} + t_{interconnect}$$

where $t_{control}$ is the delay in the next state logic, $t_{datapath}$ is the execution delay of the components in the datapath, $t_{nxt\_st\_logic}$ is the delay to compute the next state value, $t_{state\_register}$ is the hold time of the state register and $t_{interconnect}$ is the wire delays.

The architecture for the non-pipelined control architecture is shown in Figure 3(b) The total area for implementing this architecture can be given by

$$Total\_Area = A_{control} + A_{datapath} + A_{nxt\_st\_logic} + A_{state\_register} + A_{interconnect}$$

where $(A_{control})$ is the area of the control logic, $(A_{nxt\_st\_logic})$ is the area of the next-state logic, $(A_{state\_register})$ is the area of the state register and $(A_{datapath})$ is the datapath area.

2.1.1 Status Pipelined Methodology

In this model, the microactions are executed in a pipelined fashion. We can think of this model as a two stage pipeline, where the first stage performs the first two microactions ($s_{i_{set\_cntrl}}$ and $s_{i_{exec\_dp}}$) and the second stage performs the third microaction, ($s_{i_{next\_state}}$). The performance characteristic for this model is shown in Figure 3(c).

In order to achieve this pipelined performance, we now require a register on the status lines. Hence this architecture is called the *Status Pipelined Architecture.* The length of the clock cycle decreases because of the pipelining. The minimum clock period for this architecture can be given by:

$$Min\_Clk\_Period = MAX(t_{group\_1}, t_{group\_2})$$

$$t_{group\_1} = t_{control} + t_{datapath} + t_{status\_register} + t_{interconnect}$$

$$t_{group\_2} = t_{nxt\_st\_logic} + t_{state\_register} + t_{interconnect}$$

The introduction of a status register on all status lines increases the area of the design substantially. Thus the total area for this architectural style can be given by

$$Total\_Area = A_{control} + A_{datapath} + A_{nxt\_st\_logic} + A_{state\_register} + A_{interconnect} +$$

$$Num\_status * A_{status\_register}$$

2.1.2 Control and Status Pipelined Methodology

In this model, the number of pipeline stages is further increased. All the three microactions are executed in a pipelined fashion. We can think of this model as a three stage pipeline, where each stage performs one of the microactions, $s_{i_{set\_cntrl}}$, $s_{i_{exec\_dp}}$ and $s_{i_{next\_state}}$. The performance characteristic for this model is shown in Figure 3(e).

In order to achieve this pipelining performance, we have to now introduce a register on the status and control lines, (Figure 3(f)). Hence this architecture is called the *Control - Status Pipelined*

Architecture. The existence of pipeline registers on all status and control signals further enhances the performance characteristics of this architecture. The minimum clock period for this architecture can be given by:

$$Min\_Clk\_Period = MAX(t_{group\_1}, t_{group\_2}, t_{group\_3})$$

$$t_{group\_1} = t_{control} + t_{control\_reg} + t_{interconnect}$$

$$t_{group\_2} = t_{datapath} + t_{status\_reg} + t_{interconnect}$$

$$t_{group\_3} = t_{nxt\_st\_logic} + t_{state\_register} + t_{interconnect}$$

The existence of registers on all status and control lines increases the area figures for this architecture. The total area for this architectural style can be given by

$$Total\_Area = A_{control} + A_{datapath} + A_{nxt\_st\_logic} + A_{state\_register} + A_{interconnect} +$$

$$Num\_status * A_{status\_register} + Num\_control * A_{control\_register}$$

3 Impact of Control Pipelining on Operation Scheduling

When the designer imposes an architectural constraint on the design, (say by selecting one of the above control pipelining schemes) high level synthesis algorithms must be capable of working with this constraint. Since scheduling is one of the first and important phases during high level synthesis, scheduling algorithms are directly impacted by this architectural constraint imposed by the designer. In this section we discuss some of the architecture-related aspects that influences scheduling.

We will illustrate this impact on scheduling by using a simple example shown in Figure 4. This example consists of eight operations spread over three branches of a case statement. When scheduling this example, there are two important aspects that are directly related to the architectural constraint. (i) sharing of mutually exclusive operations (*operations d, g and i*) in the same state and (ii) scheduling the condition-evaluation operation and testing the results of condition-evaluation (*operation b*).

3.1 Sharing of Mutually Exclusive Operations

If the behavior description contains mutually exclusive branches (like the one shown in Figure 4) the scheduling algorithm has to make an important choice when scheduling operations on these branches. The mutually exclusive operations can either be (i) shared and hence scheduled into the same state, or (ii) not-shared and hence scheduled into two different states.

In Figure 5(a), we show the results of a scheduling algorithm that shares mutually exclusive operations in the same state. In this figure, *operations d, g* get scheduled into the same state *(state 2)* because they will never get executed simultaneously during any single execution of the process. Similarly, *operations e and h* also get scheduled into the same state *(state 3)*. This sharing of mutually exclusive operations obviously results in few states for implementing the design. For the rest of this report we refer to schedulers that can share mutually exclusive operations as *sharing-schedulers* and their schedules as *sharing-schedules.*

In Figure 5(b), we show the results of a scheduling algorithm that does *not* share mutually exclusive operations. It schedules these operations separately into individual states. For example, it schedules *operations d and g* into separate states *(i.e., states 2 and 5 respectively).* Schedulers that do not share mutually exclusive operations are called *non-sharing schedulers* and their schedules are called

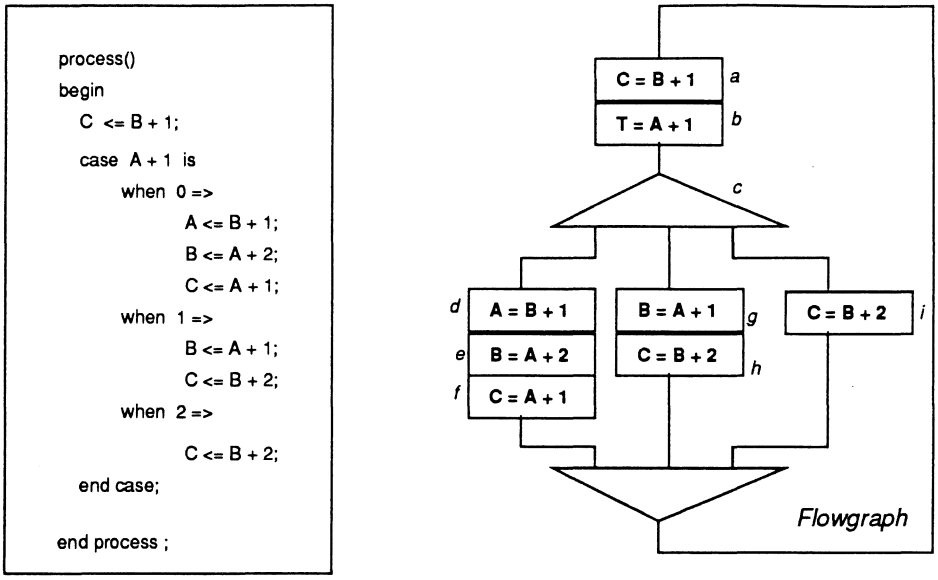

Figure 4: A Simple Example

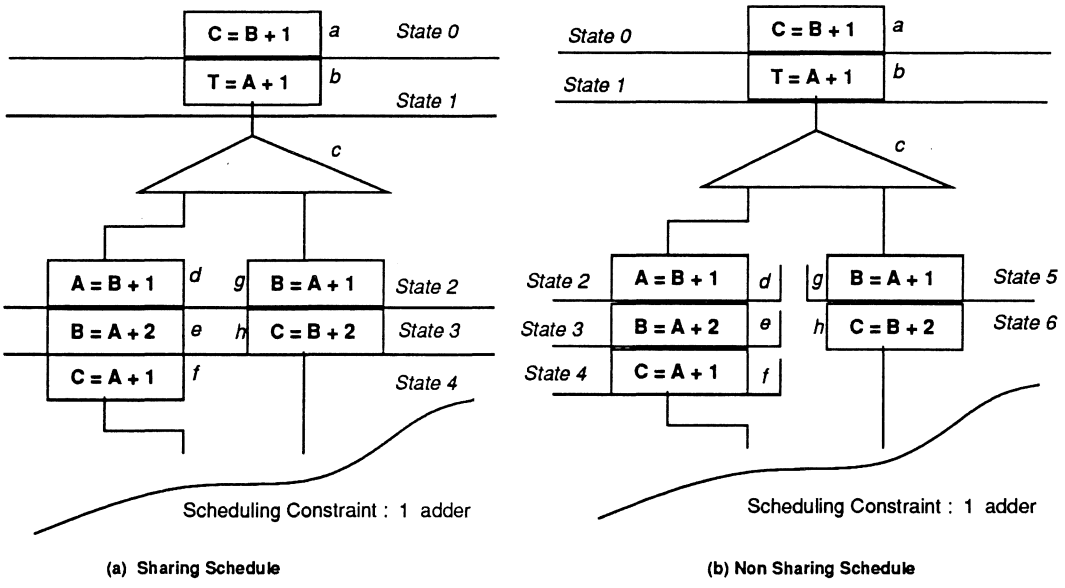

Figure 5: Mutually Exclusive Operations

non-sharing schedules. The total number of states produced by a non-sharing scheduler is obviously larger than the number of states produced by a sharing scheduler.

It is generally argued [1,3,5,9] that a sharing-schedule produced by a sharing-scheduler, is a 'superior' schedule because the total number of states are fewer. In fact many scheduling algorithms [1,5,9] spend most of their effort, trying to share as many mutually exclusive operations as possible into a single state. However, we show in the rest of this report that this is not necessarily useful. *The non-sharing schedule is better suited for some of the architectures, while the sharing-schedule is better suited for other architectures.*

3.1.1 Non pipelined Architectures

Let us now try to implement the 'superior' schedule, on a non-pipelined model (Figure 3(a)). As shown in Figure 5(a), the value of T is computed in state 1 in this schedule. This value of T cannot be stored anywhere since the architecture does not provide for a status register. However, the value of T is required in states 2 and 3, to determine which of the two shared operations have to be performed.

Since the value of T was not stored anywhere, it is impossible to determine what actions are to be performed in states 2 and 3. Therefore, we can conclude that, *it is not practical to share mutually exclusive operations in architectures that do not have a status register.*

A non-sharing schedule (Figure 5(b)), precludes this problem because the value of T is required only in state 1 to determine whether to transition to state 2 or state 5. Once this transition to one of the two states takes place, the value of T is not required anymore. Therefore the lack of a status register will not cause any problems for implementing a schedule created with a non-sharing scheduler.

Conclusion 1: *For non-pipelined architectures, it is better to use a scheduler that does not share mutually exclusive operations.*

3.1.2 Status Pipelined Architectures

Let us try to implement the 'superior' schedule on an architecture with a status pipeline register (shown in Figure 3(b) and Figure 3(c)). These architectures are more amenable for implementing the sharing-schedule because the value of T can now be stored on the status registers and used whenever desired. In our example, T can be stored into the status register in state 1 and used in states 2 and 3 to determine which of the two mutually exclusive operations need to be performed, in these states.

On the other hand, we face problems if we implement a non-sharing schedule on these architectures. Refering back to our example, the condition-evaluation is scheduled in state 1 and stored in the status register during the subsequent clock rising event. Thus the value of the status lines are not available to the controller in state 1. The controller cannot decide whether the next state is 2 or state 5. This problem can be solved by introducing a no-op (dummy) state after state 1. The status bits will be available to the controller during this dummy state and the controller can transition to either state 2 or 5 based on the values in the status bits. However, this solution creates inefficiencies, by introducing dummy states unnecessarily.

Conclusion 2: *For architectures with a pipeline register at the status bits, it is more efficient to use a sharing-scheduler.*

3.2 Condition Evaluation and Testing

All operations in a given example can be classified into two categories. (i) computation operations and (ii) condition-evaluation operations. Condition evaluation operations are used to select a path from a set of paths. In Figure 4, operation *b* evaluates the condition (I + 1), whose result determines which of the three branches have to be taken. Hence this operation is called *the condition-evaluation operation.*

The results after condition-evaluation are eventually available on the status lines (which is either pipelined or non-pipelined). The controller then tests the status lines to determine what future action is to be taken. This is called *condition-testing.* The results of condition-testing are used differently, based on the scheduler type. In Figure 5(a) the value on the status lines determines the action that is performed in a given state, while in Figure 5(b) the value on the status lines determines the next state.

There is a strong relationship between architecture specification and scheduling of condition-evaluation and condition-testing operations.

3.2.1 Non-pipelined Architectures

In non-pipelined architectures the status lines will get updated as soon as the condition- evaluation operation is scheduled. Moreover, since the status lines are not pipelined, the status bits do not get stored anywhere. Hence testing of the status conditions must occur immediately. In Figure 6(a) we show this relationship between scheduling a condition-evaluation operation and the availability of the status signals for testing.

Synthesis algorithms must take into account this relationship that exists because of the architectural specification. We have showed earlier that a non-sharing scheduler is to be used for non-pipelined architectures. Since these non-pipelined architectures lack a storage unit on the status lines, and the result of a condition-evaluation is required to determine which branch is to be taken, the condition-evaluation operation must be scheduled as the *last operation in the previous basic block.*

Let us consider our example: the first basic block has two operations, *a and b.* Operation *b* is the condition-evaluation operation. Hence, it has to be scheduled as the last operation in the first basic block for this architecture. If operation *b* were scheduled earlier (say state 0), proper branching is impossible.

From the above discussion we come to the next conclusion which is: **Conclusion 3:** *For non-pipelined architectures, the condition-evaluation operation has to be the last scheduled operation in the previous basic block*

3.2.2 Status Pipelined Architectures

For status pipelined architectures, the status signals are available as soon as the condition-evaluation operations are scheduled, but testing can only occur in the following state because of the register on the status line. Thus there is a one control-step delay between scheduling a condition-evaluation operation and the testing of the status signals. In Figure 6(b) we show this relationship between scheduling a condition-evaluation operation and the availability of the status signals for testing.

When scheduling for this architecture, the condition-evaluation must be scheduled *at least* one step before the value is required for testing. According to our *Conclusion 1*, we know that a sharing scheduler will be used for status pipelined architectures, (Figure 5(a)). Hence the condition-evaluation operation has to be scheduled accordingly. In our example, the earliest state in which we require to

test the conditions on the status signals is *state 2*. Thus the condition-evaluation operation will have to be scheduled on or before *state 1*. Hence operation *b* can be either in state 0 or state 1.

Conclusion 4: *For status-pipelined architectures, the condition-evaluation can be scheduled anywhere in the basic block.*

3.2.3 Control - Status Pipelined Architectures

For control-status pipelined architectures, the delay between scheduling the operation and the ability to test its status is at least two control steps. i.e., if the condition-evaluation operation is scheduled in state 1, the status bits can be tested only in state 3. This chronological relationships between the scheduling of condition-evaluation operations, the updating of status signals and the testing of the status signals for this architectural types is shown in Figure 6(c).

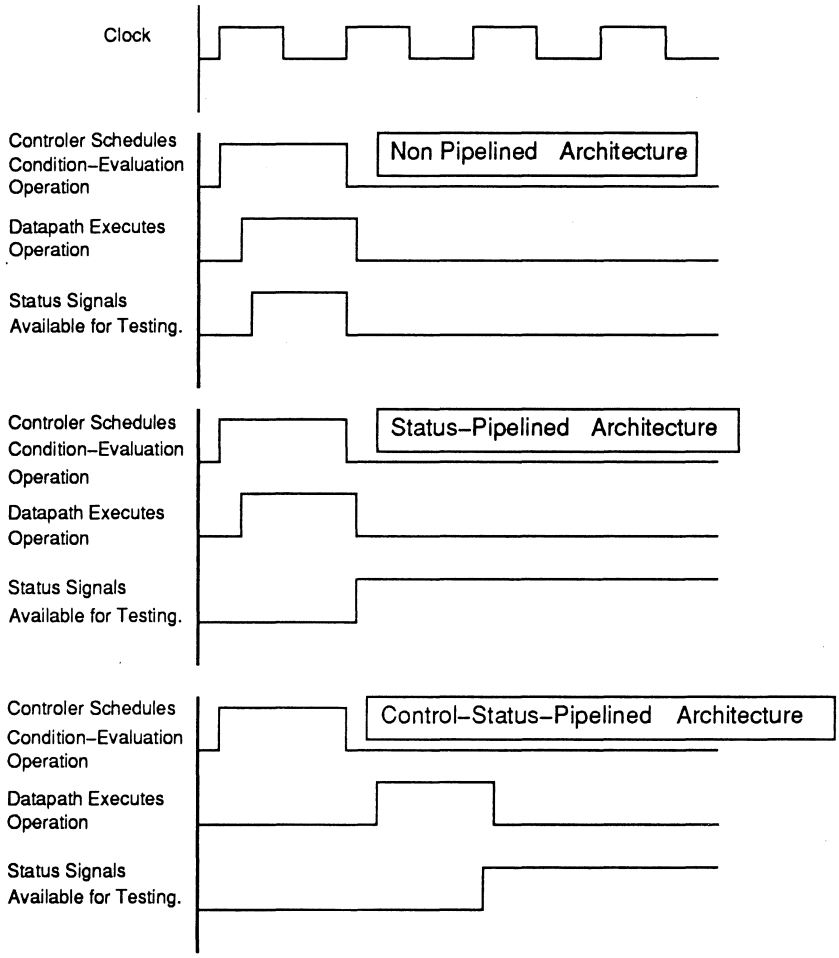

Figure 6: Condition Evaluation and Testing

When scheduling for this architecture, the condition-evaluation must be scheduled *at least two steps* before the value is required to be tested. According to our *conclusion 2*, the sharing-scheduler will be used for this architecture too. In our example, since the status bits will be tested in state 2 to determine which of the operations are to be performed, the condition-evaluation has to be scheduled in state 0.

	Scheduler Type	Condition Evaluation
NonPipelined Architecture	*NonSharing*	*Scheduled in the n'th state of a n–state basic block*
Status Pipelined Architecture	*Sharing*	*Scheduled anywhere between 1 to 'n' for a n–state basic block*
Control Status Pipelined Arch.	*Sharing*	*Scheduled anywhere between 1 to 'n –1' in the basic block*

Figure 7: Impact on Scheduling

In some cases (especially when the basic block is very small, or because of dependencies in the basic block), it may not be possible to have a state after the condition evaluation operation. In such cases it is necessary to introduce a dummy operation to make the schedule compatible with the architecture.

Conclusion 5: *In general, we can conclude that for control-status-pipelined architectures, the condition-evaluation operation has to be scheduled on or before the penultimate step in the basic block. No-op introduction may be necessary if the condition-evaluation operation has to be scheduled in the last step of the basic block.*

We have summarized all the above conclusions in Figure 7, which shows the impact of architectural specifications on the scheduling algorithms.

4 Scheduling Algorithm

The architecture based scheduling algorithm can be now derived based on our conclusions thus far (Figure 7). The input textual description for the behavior is first translated into a control dataflow graph (CDFG). This CDFG consists of a set of control nodes C = $c_1, c_2, ..c_n$. All nodes in C can be separated into two categories (i) stmt_blk nodes and (ii) control split/join nodes. Let S = $s\_1, s\_2, ..s\_n$ be the set of all stmt_blk nodes in C. Each node in S contains a dataflow graph which is an interconnected of dataflow operations.

The overall scheduling algorithm for the CDFG (C) is shown in Algorithm 1. In this algorithm, the architecture specification is obtained from the designer. Then the **architecture_based_list_scheduling** function is invoked on each stmt_blk node in the CDFG. Depending on the selected architecture type we either invoke a sharing scheduler or a nonsharing scheduler on the global CDFG.

The architecture_based_list_scheduling algorithm is based on 'mobility' similar to the algorithms published in [10]. Here the nodes are prioritized based on a factor called 'mobility' and are then assigned to states based on this factor. However since the algorithm has to accommodate a wider range of architectures, we have to incorporate our conclusions in Figure 7 during the scheduling process.

The details of the function *architecture_based_list_scheduling* are shown in Function 1(a). This function first computes the mobilities of all the nodes in the stmt_blk and determines which of the operation is a condition-evaluation operation. If a non_pipelined architectural style is required, the function schedules the nodes using the **schedule_nodes** function and then modifies the schedule to ensure that the condition-evaluation operation is postponed to be the last state in the stmt_block. For

Algorithm 1: Scheduling_with_architectural_constraints

```
begin
   arch_type = get_architectural_specification()

   foreach s_i ∈ S
      architecture_based_list_scheduling(s_i, arch_type);
   end for

   if arch_type == non_pipelined
      invoke_nonsharing_scheduler(CF);
   else
      invoke_sharing_scheduler(CF);
   end if

end
```

Function 1(a) architecture_based_list_scheduling (s_i, arch_type)

```
begin
   Let mobilities_list be a set of mobility values for all operations
   mobilities_list = calculate_nodes_mobility(s_i);
   cond_op = determine_cond_eval_op(s_i);

   switch arch_type
      case non_pipelined_arch:
         last_state = schedule_nodes(s_i, mobilities_list);
         modify_schedule(cond_op, last_state);
      case control_status_pipelined_arch:
         decrease_mobility_by_one(cond_op);
         last_state = schedule_nodes(s_i, mobilities_list);
         add_no_ops_if_required(s_i);
      case status_pipelined_arch:
         last_state = schedule_nodes(s_i, mobilities_list);
   end switch
end
```

a control-status-pipelined architecture, the mobility of the condition-evaluation operation is decreased by one and then the scheduling is done. This forces the **schedule_nodes** function to try and schedule the condition-evaluation operation at least one step before the last. If the scheduling algorithm was not succesful in making the condition-evaluation to be one step before the last one, then an additional no_op state is introduced as the last state in the stmt_blk. For a status-pipelined architecture a simple mobility-based algorithm will suffice, since there are no special requirements.

The details of the nonsharing scheduler is given in Function 1(b). In this algorithm the condition vectors are computed using the function **compute_condition_vectors**, for each stmt_blk node in the CDFG. Computing condition vectors is a two step process. In the first step, all the control variables in the description are determined. (In VSS, all control variables are named as 'T' followed by a number. Thus T1, T2 ... are typical control variable names). The number of control variables in the CDFG determines the size of the condition vector. In the second step the actual value of the condition vectors are calculated for each stmt_blk. As an example if a stmt_blk will be executed under the conditions T1 is 1, T2 is 0 and T3 is a 'dont_care' then the condition vector for that block is 10-. (where '-' indicates a 'dont-care').

After computing the condition vectors, the control_queue is initialized with the first stmt_blk of the design. During each iteration the top node of this queue *c* is removed and scheduled using the routine **schedule_sequentially**. The variable *start_state* keeps track of the starting value for the current stmt_blk. After scheduling the stmt_blk state transitions are added to each successor stmt_blk maintained in the SUCC queue. Finally the nodes in the SUCC queue are appended to the control_queue and the next iteration continues.

The sharing scheduler is quite similar to the nonsharing one and the details are shown in Function 1(c). Since all the nodes in the control_queue have to share the states, the entire control_queue is passed to the **schedule_sequentially** routine. After scheduling all the blocks in the control_queue the maximum state value is returned. State transitions are added from the last_state of each element in the control_queue to the (max_state + 1). The control_queue is then replaced with the elements in the SUCC_ALL queue and the iterations continue.

The results of running the above scheduling algorithm on the example shown earlier in Figure 4 is shown in the following figure, (Figure 8).

In this figure, the first row contains the results of architecture-based list scheduling on each of the stmt-blks in the design. There are four stmt_blks in our simple example. The first stmt_blk contains a condition-evaluation operation (operation *b*) which is scheduled according to the conclusions in Figure 7. As shown in Figure 8, the condition-evaluation is scheduled as the penultimate operation in control-status pipelined architecture, while it is the last operation in the other two architectures.

In the second row (Figure 8) we show the results of applying the corresponding global control scheduling algorithm. Here the non-sharing scheduling is applied for the non-pipelined architecture and the sharing scheduler is applied for the other two architectures. The sharing scheduler requires only five states while the non-sharing scheduler requires eight states to complete the schedule.

In the third row of Figure 8, we show the final control unit and datapath achieved for all the three architectures. While the datapath is very similar for the three cases the control unit varies widely for all these three architectures. The non-pipelined architecture has extra transitions since there are three possible next states from state s1, while the remaining two architectures have extra control lines to control the status registers.

Function 1(b) invoke_nonsharing_scheduler (C)

```
begin
   foreach node s_1, s_2, ...s_n
      compute_condition_vectors( s_i);
   end for
   control_queue = first_statement_block;

   while control_queue is not empty
      Let c = first_element(control_queue);
      Let {SUCC = succ_1, succ_2...succ_n} be the successor stmt_blks of (c):
      start_state = schedule_sequentially(start_state, c)

      foreach element succ_i ∈ SUCC where i = 1 to n
         start_state = add_state_transition(start_state, start_state + i, cv(c), cv(succ_i))
      end for
      control_queue = append_queues(control_queue, SUCC);

   end while
end
```

Function 1.1(c) invoke_sharing_scheduler (C)

```
begin
   start_state = 0;
   foreach node s_1, s_2, ...s_n
      compute_condition_vectors( s_i);
   end for
   control_queue = first_statement_block;
   Let {SUCC_ALL = succ_1, succ_2...succ_n} be the successor
        stmt_blks of all the nodes in the control queuec):
         max_end_state = schedule_sequentially(start_state, control_queue)

         foreach element c_iin control_queue
            add_state_transition(last_state(c_i), max_end_state + 1, cv(c_i))
         end for
         start_state = max_end_state
      control_queue = SUCC;
end
```

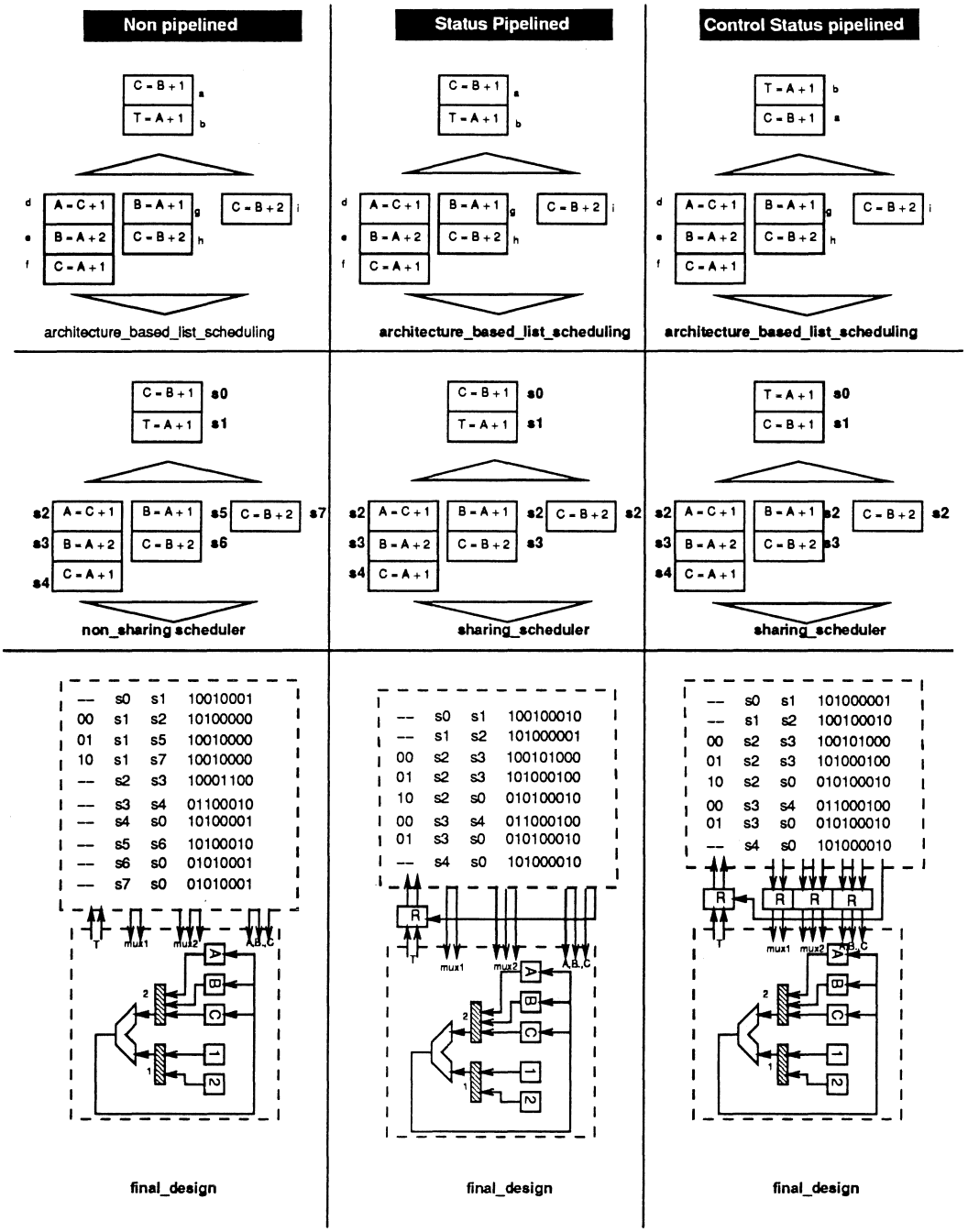

Figure 8: Scheduling Results

5 Experiments and Results

We have incorporated our architecture based synthesis methodology into our VHDL Synthesis System (VSS), [6,12] which is fully implemented in 'C' language running on a Sparc Workstation. We have tested this architecture based synthesis methodology on a wide range of examples.

In this section we use five different examples to show the relationship between architectural-specification and synthesis. These examples are (i) our simple example introduced earlier. This example is small enough to provide an good understanding of possible architectural tradeoffs (ii) a clock division circuit, (iii) a timer circuit (iv) a counter circuit with a specific counting sequence [13] (v) An example taken from a recent conference publication[5].

The synthesis process was invoked for each of the three architecture types. After synthesizing the datapath with the appropriate type of scheduling algorithm, the control part was synthesized using MUSTANG [15] for state encoding and MIS[16] for logic optimization.

For these examples, we compute the actual area of the synthesized design based on the number of transistors used. In order to compute the number of transistors for the design we use the following table.

Component	B/w	#In	Equation
ADD–SUB	b		34 * (n–1) + 12
ALU	b		35*b
MUX	b	i	4*i*b + 4 * b
REG	b		24
INV			2
NAND		i	2*i
NOR		i	2*i
OR		i	2*i + 2
AND		i	2*i + 2
MULT		i	6*b*b + 14*(b–1) + 34*(b–1)(b–2)

Figure 9: Transistor Equations

5.1 Our example

We have already discussed the synthesis process for this example in the previous section. Let us now show the synthesis results for all the architectures.

The list of components for the three architectures is shown in Figure 10. The first column shows the name of all the components used in the design. The second column shows a bit-width and number of inputs for each components. The last three columns actually indicate the number of components of that particular type that were used in the final design. In this particular example, three one-bit registers were used for the non-pipelined architecture, while only two of them were required for the status-pipelined architecture.

The state based performance characteristics like the maximum number of states, the min/max/total number of states for a single execution path is shown in the lower portion of the figure.

In this particular case, the datapath turned out to be the same in all the cases. The differences in area were mainly contributed by pipeline registers. The total register-bits used to implement the designs varies significantly for the architectures (6 for Nonpipelined, 8 for StatusPipelined and 16

for Control-Status Pipelined). This difference is caused by the pipelining of control and status lines. It is clear from Figure 10 that the area relationship is: $Area_{non\_pipelined} < Area_{status\_pipelined} < Area_{control-status-pipelined}$

This a wide range of area/delay tradeoffs are possible by examining different architecture styles. Also, in this particular example the schedule with the maximum number of states is the most useful since the corresponding architecture produces the least area.

Component		Non Pipelined Architecture	Status Pipelined Architecture	Con/St. Pipelined Architecture
Name	bw/In			
ALU	2	1	1	1
MUX	3	1	1	1
MUX	2	1	1	1
REG	1	3	3	11
REG	2	0	1	1
REG	3	1	1	1
INV	1	11	8	10
NAND	2	7	13	5
NAND	3	5	6	6
NAND	4	1	2	1
AND	2	1	0	0
AND	3	0	0	0
AND	4	0	0	0
NOR	2	8	0	1
NOR	3	2	0	2
NOR	4	3	0	1
OR	2	2	5	3
OR	3	0	0	0
OR	4	0	0	0
TOTAL NUMBER OF TRANSISTORS		**420**	**434**	**612**

PERFORMANCE	Min States	3	3	3
	Max States	5	5	5
	Total States	8	5	5

Figure 10: Results - Simple Example

5.2 Timer Circuit

The timer circuit contains two cascaded counters. During a given interval the counters are decremented continually, The counters can initially be loaded with two values n and m respectively. Since the

counters are cascaded the first counter decrements by n before the second counter decrements by *1* After both the counters reach *0* an output pulse is sent.

The synthesis results for all the three architectures is shown in the Figure 11. Here again the important differences in the area is caused by the pipeline registers and the control logic. The state register for the non-pipelined register is larger by one bit because of the larger number of states, but this is compensated for, by the extra pipeline registers in the other architectures.

Component		Non Pipelined Architecture	Status Pipelined Architecture	Con/St. Pipelined Architecture
Name	bw/In			
ALU	16	1	1	1
ALU	16	1	1	1
MUX	16/2	3	3	3
MUX	8/3	1	1	1
MUX	16/3	1	1	1
MUX	16/5	1	1	1
MUX	1/2	4	4	4
REG	8	3	3	3
REG	16	2	2	2
REG	1	12	23	76
INV	1	33	37	47
NAND	2	26	15	29
NAND	3	16	11	19
NAND	4	5	7	8
AND	2	4	3	2
AND	3	0	0	0
AND	4	0	0	0
NOR	2	23	12	13
NOR	3	11	18	21
NOR	4	3	4	9
OR	2	1	2	6
OR	3	0	0	0
OR	4	0	0	0
Number of Trans		**4624**	**4844**	**6328**
PERFORMANCE	Min States	2	2	4
	Max States	16	16	28
	Total States	26	16	28

Figure 11: Results - Timer

5.3 Clock Division Circuit

The clock division circuit is used to divide the input clock frequency f_{in} by the ratio of N by M, where N and M are the input ports to the circuit. The output frequency can be characterized by the equation $f_{out} = (\mathrm{N}/\mathrm{M})\ f_{in}$.

The synthesis results for all the three architectures is shown in the Figure 12. The tradeoffs and conclusions are very similar to those for the timer circuit.

Component		Non Pipelined Architecture	Status Pipelined Architecture	Con/St. Pipelined Architecture
Name	bw/In			
ALU	23	1	1	1
ALU	23	1	1	1
MUX	23/2	7	7	7
MUX	23/3	1	1	1
MUX	1/2	1	2	2
REG	23	4	4	4
REG	1	6	12	51
INV	1	21	23	25
NAND	2	16	5	6
NAND	3	2	5	11
NAND	4	2	6	5
AND	2	0	0	1
AND	3	0	0	0
AND	4	0	0	0
NOR	2	6	14	14
NOR	3	3	6	9
NOR	4	2	2	5
OR	2	13	1	2
OR	3	0	0	0
OR	4	0	0	0
Total Number of Transistors		**6499**	**6631**	**7645**
PERFORMANCE	Min States	8	11	17
	Max States	12	12	18
	Total States	14	12	18

Figure 12: Results - Clock Division

5.4 Rockwell Counter

The Rockwell Counter is one of the industrial benchmarks that has been used to test the performance of our synthesis system. The counter has a start count of 0 and a terminal count of 3327. During each strobe the counter increases by 208. If the count is greater than the terminal count the counter will start at the previous beginning of the count plus 26. If this counter initial value is greater than 207 then the initial value will be previous initial value plus 1. More details of the counter can be found in [14,15].

The synthesis results for all the three architectures is shown in the Figure 13.

Component		Non Pipelined Architecture	Status Pipelined Architecture	Con/St. Pipelined Architecture
Name	bw/In			
ALU	12	1	1	1
ALU	12	1	1	1
MUX	12/3	2	2	2
MUX	12/4	1	1	1
REG	12	2	2	2
REG	1	4	8	31
INV	1	13	18	21
NAND	2	6	3	8
NAND	3	3	2	4
NAND	4	0	3	7
AND	2	0	1	2
AND	3	0	0	0
AND	4	0	0	0
NOR	2	14	7	9
NOR	3	7	6	5
NOR	4	1	2	3
OR	2	3	1	4
OR	3	0	0	0
OR	4	0	0	0
Total Number of Transistors		**2294**	**2374**	**3030**
PERFORMANCE	Min States	3	3	4
	Max States	7	7	12
	Total States	12	7	12

Figure 13: Results - Rockwell Counter

5.5 Kim's Example

This is a random dataflow graph used in [5] to discuss a scheduling approach that tries to minimize the states, by using various transformation techniques. The example consists of a couple of branch statements with long calculation chains of addition and multiplication.

The synthesis results for all the three architectures is shown in the Figure 14.

Component Name	bw/In	Non Pipelined Architecture	Status Pipelined Architecture	Con/St. Pipelined Architecture
ALU	16	1	1	1
ALU	10	1	1	1
MULT	10	1	1	1
MUX	10/8	1	1	1
MUX	10/6	1	1	1
MUX	16/2	2	2	2
MUX	11/2	1	1	1
REG	11	12	12	12
REG	1	5	6	45
INV	1	30	31	31
NAND	2	18	19	19
NAND	3	10	7	8
NAND	4	8	3	5
AND	2	2	5	3
AND	3	0	0	0
AND	4	0	0	0
NOR	2	18	13	24
NOR	3	19	22	20
NOR	4	3	8	6
OR	2	12	4	11
OR	3	0	0	0
OR	4	0	0	0
Total Number of Transistors		**9040**	**9020**	**10024**
PERFORMANCE	Min States	13	13	13
	Max States	15	15	17
	Total States	28	15	17

Figure 14: Results - Kim's Example

5.6 Summary of Results

In the following table we show the summary of the results. In this table, the first two columns shows the design name and the selected architectural style. The third column shows the total number of transistors that were used by VSS to implement the design. The next three columns show the total

number of states in the design, and the paths with min and max lengths.

		Transistors	TotalStates	Min States	Max States
Simple Example	NonPipelined Arch.	420	8	3	5
	Status Pipelined	434	5	3	5
	Control Status Pipelined	612	5	3	5
Timer	NonPipelined Arch.	4624	26	2	16
	Status Pipelined	4844	16	2	16
	Control Status Pipelined	6328	28	4	28
Clock Division	NonPipelined Arch.	6499	14	8	12
	Status Pipelined	6631	12	11	12
	Control Status Pipelined	7645	18	17	18
Counter	NonPipelined Arch.	2294	12	3	7
	Status Pipelined	2374	7	3	7
	Control Status Pipelined	3030	12	4	12
ICCAD example	NonPipelined Arch.	9040	28	13	15
	Status Pipelined	9020	15	13	15
	Control Status Pipelined	10024	17	13	17

Figure 15: Summary of Results

6 Conclusions

The research presented in this paper clearly shows that architectural constraints play a very important role in the synthesis process. We have defined a new methodology that would incorporate such architectural constraints during the scheduling process. Based on the architectural constraint provided, the resultant design varies in area and performance significantly. From our experiments with this methodology we conclude the following:

- Synthesis tools should be flexible enough to handle a wide range of architectures as required by designers. Tools intended for a single architecture may not be useful for other architectures and hence may not really be usable.
- Architectural constraints are more useful than typical synthesis constraints like number of functional units, because typically designs do not contain more than one functional unit per operation type.
- Scheduling for the minimum number of states, does not necessarily produce a design of acceptable quality. Schedules with the minimum number of states may not be implementable in many architectures.

7 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #91-DJ-146). We are grateful for their support. We would also like to thank Viraphol Chaiyakul for his suggestions and useful discussions during the course of this project.

8 References

[1] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Trans. CAD*, Vol.10, no.1, pp.85-93, Jan. 1991.

[2] R. Camposano and W. Rosenstiel, "Synthesizing Circuits from Behavioral Specifications," *IEEE Trans. CAD*, vol.8, no.2, pp.171-180, Feb. 1989.

[3] R. Camposano and W. Wolf, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, 1991.

[4] D.D.Gajski, N. Dutt, A. Wu, S. Lin, *High-Level Synthesis*, Kluwer Academic Publishers, 1991.

[5] T. Kim, J.W.S. Liu, and C.L. Liu, "A Scheduling Algorithm For Conditional Resource Sharing," *Proc. ICCAD'91*, pp.84-87, 1991.

[6] J.S. Lis and D.D. Gajski, "Synthesis from VHDL," *Proc. IEEE Int. Conf. on Computer Design'88*, pp.378-381, 1988.

[7] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," IEEE Trans. CAD, vol.8, no.6, pp.661-679, Jun. 1989.

[8] C.-J. Tseng, R.W. Wei, S.G. Rothweiler, M. Tong and A.K. Bose, "Bridge: A Versatile Behavioral Synthesis System," *Proc. 25th DAC.*, pp.415-420, 1988.

[9] K. Wakabayashi and T. Yoshimura, "A Resource Sharing Control Synthesis Method for Conditional Branches," *Proc. ICCAD'89*, pp. 62-65, 1989.

[10] B. Pangrle and D. D. Gajski, "State Synthesis and Connectivity Binding for Mircoarchitectural Compilation." *Proc. ICCAD'86.*

[11] A.Parker, J. Pizzaro and M.Mlinar, "MAHA: A Program for Datapath Synthesis", *Proc. DAC'86.*

[12] J.S. Lis and D.D. Gajski, "Behavioral Synthesis from VHDL Using Structured Modeling," *Technical Report 91-05*, University of California, Irvine.

[13] D.Gajski, J.Lis, N.VanderZanden and A.Wu, "Synthesis from VHDL: Rockwell-Counter Case Study," *Technical Report 90-09*, University of California, Irvine.

[14] N. Dutt, "GENUS: A Generic Component Library for High Level Synthesis," *Technical Report 88-22*, University of California, Irvine.

[15] S.Devadas, H-k.Ma, A.R.Newton and A. Sangiovanni-Vincentelli "MUSTANG: State Assignment of Finite State Machines Targeting Multi-Level Logic Implementations", *IEEE Trans. CAD*, Dec '88.

[16] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang "MIS: A multiple level logic optimization system", *IEEE Trans. CAD*, Nov '87.

[17] U. Prabhu and B.M. Pangrle, "Superpipelined Control and Data Path Synthesis" *DAC Proceedings*, June '92.